
SuSE LINUX AG
THE LINUX EXPERTS
SuSE

Evolution of Linux towards clustering

Operating Systems, Tools and Methods for High Performance Computing on Linux Clusters

EDF R&D Clamart (France)

7th October 2003

SuSE LINUX AG

THE LINUX EXPERTS

Andrea Arcangeli

andrea@suse.de

SuSE Kernel Developer

Abstract

- The presentation will address some of the recent innovations in the linux kernel 2.6 and how they can affect various workloads, especially the ones related to clustering
- This should be most interesting for engeneers planning to deploy the 2.6 kernel in clusters
- Future cluster related kernel features will be considered too

Linux Kernel trees

- Stable 2.4.22 (Marcelo Tosatti)
  - 2.4.23-pre6 (Marcelo Tosatti)
    - 2.4.23pre6aa3 (Andrea Arcangeli)
    - 2.4.22-ac4 (Alan Cox)
- Unstable 2.5.x is closed
- Beta 2.6.x testing
  - 2.6.0-test6 (Linus Torvalds)
    - 2.6.0-test6-mm4 (Andrew Morton)
    - 2.6.0-test6-mjb1 (Martin J. Bligh)
    - 2.6.0-test1-ac3 (Alan Cox)

Lots of new features in 2.6

- Cache writeback at the pagecache layer
- BIO – new I/O entity, allow large I/O
- Asynchronous I/O
- TSO – TCP Segment Offload
- O(1) scheduler
- Pluggable I/O scheduler (deadline/as/CFQ)
- Hugetlbfs providing bigpages
- Scheduler Kernel Preemptive
- RMAP technique used to unmap address space during paging
- NPTL support
- HZ boosted to 1000
- epoll.... and lots lots more...

Many of these have been...

- ... already deployed into production in 2.4 via backports (this is the case in various enterprise server distributions out there that needed the best performance and scalability in production ASAP)
  - The most obvious example is the O(1) sched
- This is why we have 2.4 kernels in production that for various benchmarks scales and perform almost as well as 2.6
- But 2.6 is capable of things 2.4 will never do, the writeback cache rewrite and the bio, being two of the most obvious examples
  - Those important 2.6 features are not self contained, they spread all over the kernel in drivers/flesystems in non trivial ways

Almost all these new features...

- ... affects cluster applications, some directly, some indirectly
- Some helps, some may hurt a little
- Remember some of the major advances of the 2.6 kernel have been achieved in terms of SMP scalability and in terms of desktop responsiveness compared to the 2.4 kernel
- Lowest possible latency (and even maximal possible scalability, though it's not really the case here for 2.6), normally imply not the best possible throughput
- 2.6 defaults seems a very good tradeoff but you may really want to tune it for the best throughput for computing in clusters

What's not in 2.6 yet

- Some sort of transparent process migration
- FS with DSM providing a coherent cache
- Page coloring (not significant for x86)
- Different implementations exist in form of external patches (openmosix, openSSI, bproc, etc..)
- This is not only about building *massively parallel processors*, this is also about environments with tons of idle desktop machines with fast interconnects
- SCHED_IDLE can recycle the cpu cycles

Ideas about process migration

- Interconnects getting faster and cheaper
- Example: migrating gcc can be technically implemented to generate not much more interconnect network overhead than running a gcc executable from a NFS mount, with data as well in the NFS filesystem
- Maximum remote caching is the key

Process migration security

- Security implications in stealing cpu cycles from random machines exists on both sides:
  - The binaries and the data payload will be readable by the nodeowner (crypto can make it harder but the private keys will have to be somehow present on the client, it's basically as secure as DVD decryption, which mean every smart teenager will always be technically able to extract the private key if he really wants to)
  - The nodeowner will run unknown bytecode
- The first problem is unfixable
- The latter problem depends on the kernel not to have exploitable holes keeping in mind that
  - A local DoS would become a remote DoS

How the fs cache works

- Every time an application reads or writes to a file using the read/write syscalls or by mmapping the file in the address space, some piece of ram (usually in PAGE_SIZE units, so a "page") is allocated.
- This "page" is then indexed so - at a later time – we can find this page just allocated and indexed in the cache.

Cache Benefits for fs Reads

- The whole point of the cache for reads, is to avoid hitting the disk multiple times, if the same offset of the same file is being read multiple times
- Secondly the cache abstraction allows us to generate readahead (we pre-fill the cache so future reads won't need to wait for I/O to receive the data, and more important to build big contigous scsi commands to the disk that will be served with a single DMA, this is a must to generate high performance)
- the cache could be pure memory-bus overhead too (fix with O_DIRECT or RAWIO)

Cache Benefits for fs Writes

- In the general case with writes we don't care about the previous contents in the files, we would overwrite it anyways, so the cache for writes is useful for a different reason:
  - it allows writes to be asynchronous
  - secondly the cache can also avoid some write-I/O because multiple writes to the cache may result in a single I/O-write to the disk
    - For example if two writes at the same inode offset happens with a very short intermediate delay

Async buffered cache writes

- Like for reads, we allocate or we find the page in the cache.
- Then we copy the contents of the userspace buffer into the cache and we mark it dirty.
- Then we give a timeout of 30 seconds to the dirty cache when it become dirty for the first time.
- Every 5 seconds a kernel thread (called pdflush in 2.5/2.6 and kupdate in 2.4) checks the timeout of the dirty cache, and it flushes to disk the dirty cache asynchronously if needed marking it clean at the same time

Write throttling

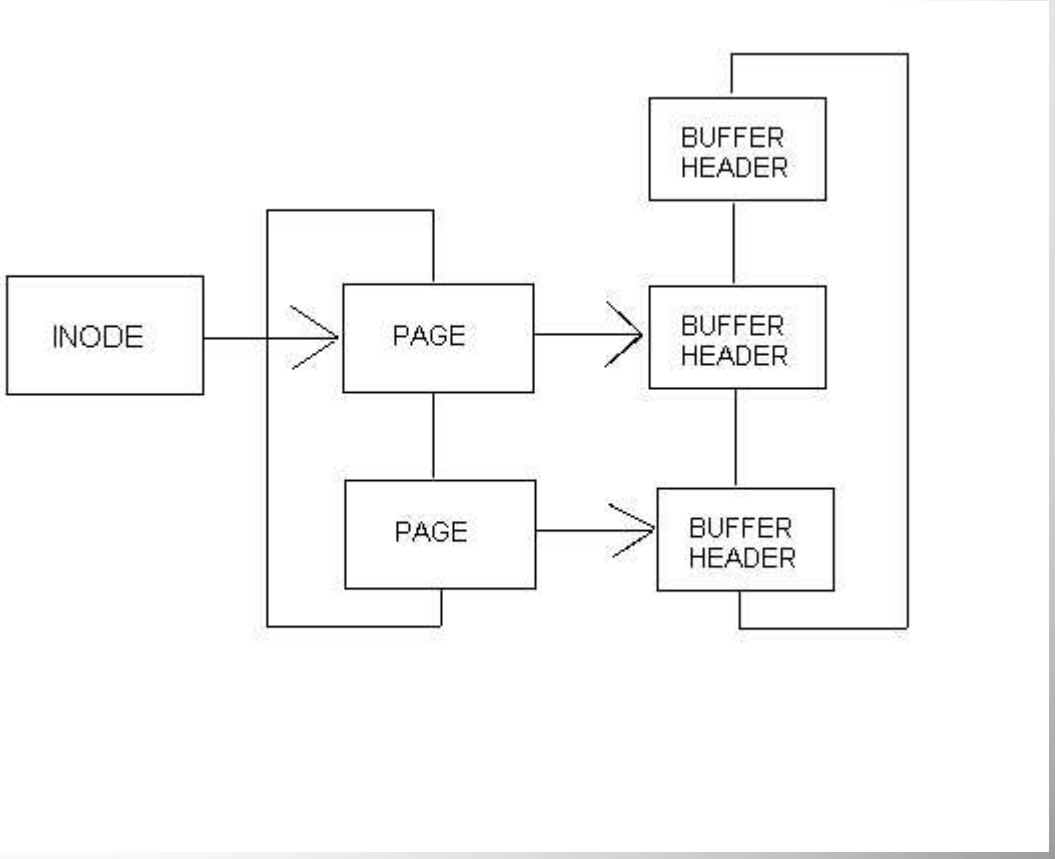
- Write intensive applications instead can still become synchronous because while we generate dirty pages, we also have check if a too large part of the vm become dirty, in such case we start writing out stuff synchronously before returning from the write operation, this is called "write throttling" and it is fundamental to avoid filling the whole vm with dirty "not immediatly freeable" pages.
- This vm-synchronous-dirty-level is also managed by a kernel daemon with an hysteresis algorithm (tunable again with the same bdflush sysctl).

2.5 writeback cache

- 2.4 kernels are used to keep track of dirty cache to flush asynchronously using a linked list of buffer-headers (aka BUF_DIRTY), that maps some memory to the physical block in some blkdev
- When it's time to flush the cache we completly lost track of its logical form.
- In 2.5 we use logical pages attached to inodes to flush dirty data, this also allows coalescing of multiple pages into a single bio submitted to the I/O layer, if the file is not fragmented on disk.

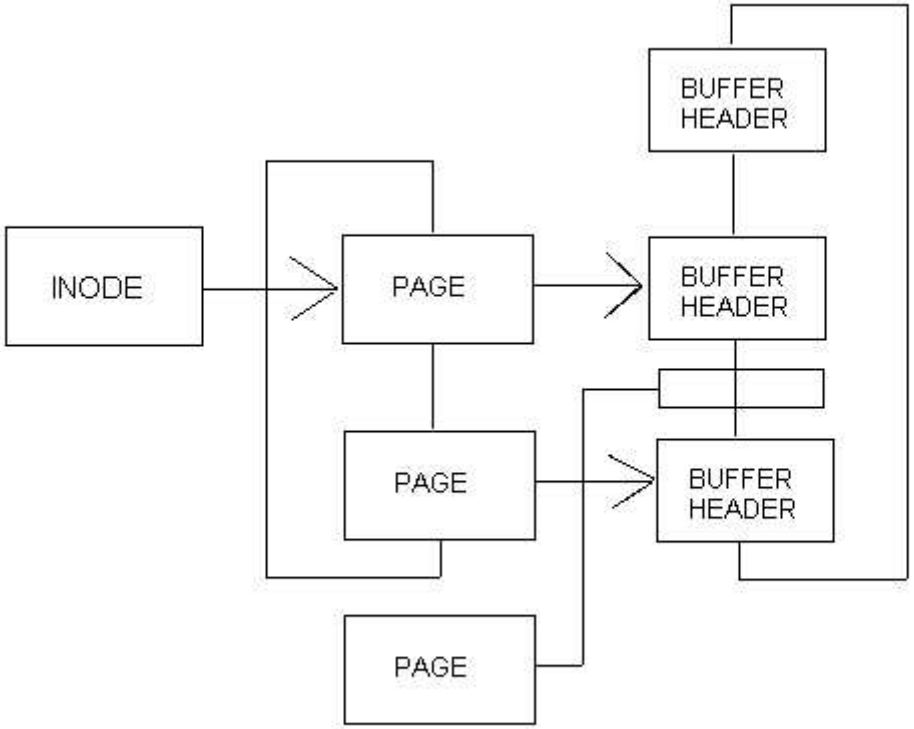
2.5 writeback cache

- Assume there are two pages dirty and they belongs to the same inode



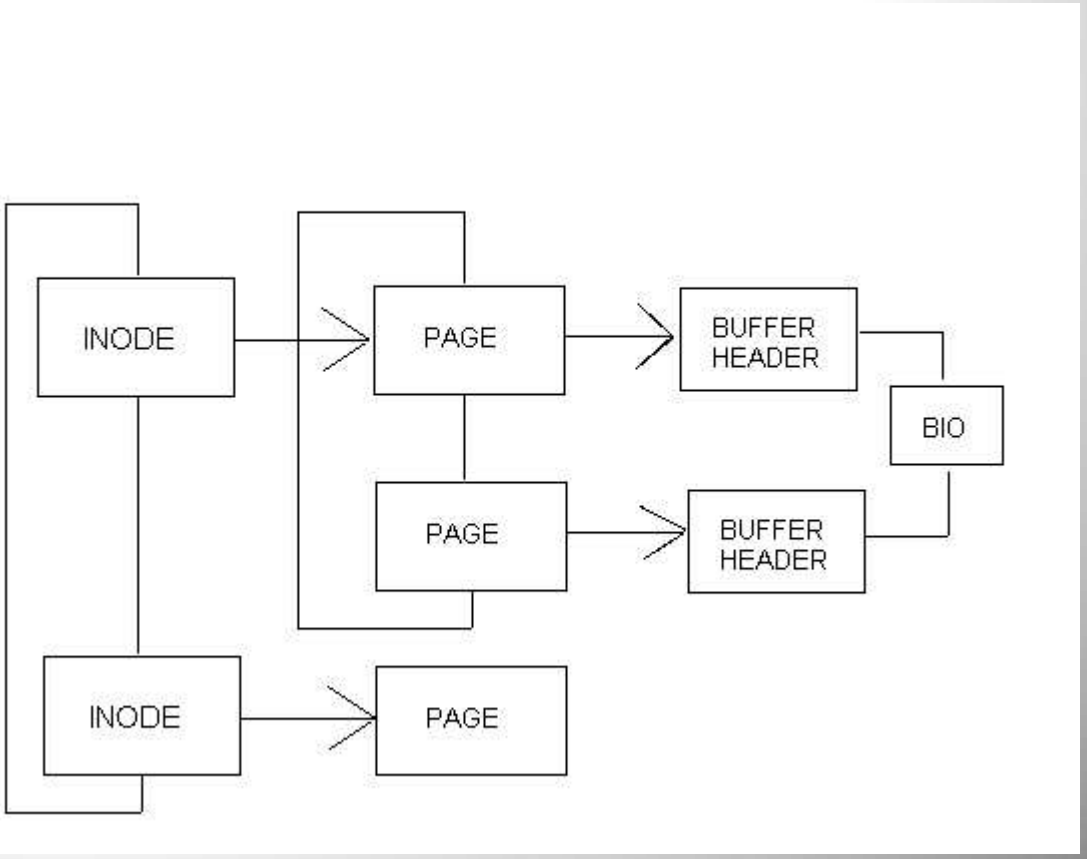
2.5 writeback cache

- But at some point some other page from other inodes is market dirty too and it gets queued in BUF_DIRTY.



2.5 writeback cache

- The new design allow us to coalesce at best all the pages from the same inodes (so probably contigous).



BIO

- Big help for:
  - Filesystems coalescing more than one page of data contigous on disk
  - O_DIRECT (when the data is contigous)
  - RAWIO (especially for large buffers)

AIO (asynchronous IO)

- Available in 2.5 (future 2.6) and 2.4.20rc2aa1 with the same kernel API
- Allows applications like databases to post read/writes and to never block
- Those apps definitely don't want to wait read/write(2) to return before they can do something else with the machine
- Current 2.4 workaround is to use threads, but context switches, task structures, message passing to other task are more costly than a true AIO that will avoid all such overhead
- Signal driven I/O completion notification is not yet available (completion ports)

SuSE LINUX AG
THE LINUX EXPERTS

Bigpages/largepages/hugepages /hugetlbfs

- x86 and other archs provides multiple PAGE_SIZEs
- PAE enabled (64G or x86-64)
  - 2M pages
- PAE disabled (4G)
  - 4M pages
- If a single tlb entry can cache more than 4k (usual PAGE_SIZE), 10 tlb entries will be able to cache an amount of VM larger than 40k: they will be able to cache up to 40M!
- Caching more VM translations into the TLB cache means less overhead in the pagetables

Bigpages/largepages/hugepages /hugetlbfs

- Very useful for number crunching too (if working with large datasets, which is a realistic scenario for some cluster workload)
- At the moment it's not provided via anonymous memory (i.e. malloc()), so a temporary file in the hugetlbfs has to be created for this (it can be deleted immediately after the mapping has been established with mmap(2))

HZ = 1000

- This will hurt the performance of the clusters doing pure userspace computations
- The slowdown for a kernel compile [from cache] (w/o altering the cacheline or the scheduler behaviour) is estimated at around 1%
- HZ=1000 doesn't help that much to make the system more responsive, because the scheduler timeslices are not affected by HZ
- We should differentiate between desktop and server/cluster environments
- We sure want HZ=100 or even less (HZ=50) for number crunching setups

HZ = 1000

- On the desktops we don't only want HZ=1000: we also want to trim the timeslices down of an order of 10, to allow the rescheduling to happen 10 times more frequently, to ***guarantee*** way more than 50 reschedules per second
- In 2.4.23pre6aa2 and in SL9, the '*desktop*' parameter will tune the scheduler to reschedule 10 times more frequently than w/o it, and at the same time it will boost HZ to 1000 dynamically
- HZ=50 can also be used with the dynamic-hz patch applied

Dynamic-hz in 2.6

- After porting dynamic-hz to 2.6, we'll cover the needs of the clusters too
- Then you may want to experiment with HZ=50

- In the meantime you can simply set HZ back to 100 to get the bit of performance back, like in some of the unofficial kernel trees

RMAP

- Another source of overhead compared to 2.4 is rmap:
  - Slowdown in
    - Page faults
    - munmap
    - Fork
  - Lots of zone-normal allocated in rmap data structures
  - (theoretical payoff during heavy paging)
- Objrmap seems to solve lots of this, despite it introduces some complexity problem

Is RMAP worthwhile?

- With today's hardware I never seen the system load being much different than ~zero during heavy swapping
- During heavy swapping most workloads become I/O dominated
- There's not much cpu to save with rmap with current common hardware
- Even if rmap would reduce the system load associated with swapping of 90%, that would be still less than 1% of the real time of the whole workload so hardly visible, while the overhead in the fast paths is definitely measurable

Page coloring

- Available as a patch for 2.4 and 2.2
- In the 2.4 patch has various problems, the engine is good, but the callers of the engine are not doing perfect static page coloring
- Ideally should be selectable per-process
- Kernel allocations should remain a dynamic page coloring
- Should allow strong and weak coloring, where strong means shrinking the cache in order to get the right color (number crunchers want the right color no matter what)
- 2.2-aa achieves most of this

O1 scheduler

- Evaluate which is the next task to reschedule during the task wakeup, not during the context switch
- Do it per-cpu with a per-cpu lock, and load balance once in a while
- High performance with an huge number of tasks running, in particular in SMP thanks to the improved scalability
- Now being improved further for higher desktop responsiveness in 2.6.0-test6, dubious in terms of throughput though
- Some version is very HT aware too

preempt/lowlatency

- Kernel compile time config option
- Implicitly disables preemption in front of the spinlocks
- Must disable preemption before accessing per-cpu data structures and before spinning
  - in_interrupt()
  - smp_processor_id()
  - spin_lock implicitly disables preemption
  - ...
- Adds a significant complexity (and a number of branches) to the kernel with the object of reducing scheduler latencies
- Number crunchers don't want/need it

preempt/lowlatency

- Preempt is mostly interesting for realtime digital signal processing where the mean latency matters
- Almost doesn't matter for playback, playback cares about the worst case latency
- Lowlatency is more important than preempt
- Preempt needs lowlatency too, preempt cannot schedule inside a spinlock region
  - Most CPU bound kernel loops tends to be inside spinlock regions, so preempt without lowlatency special care would not be enough
  - lowlatency is normally enough and low overhead

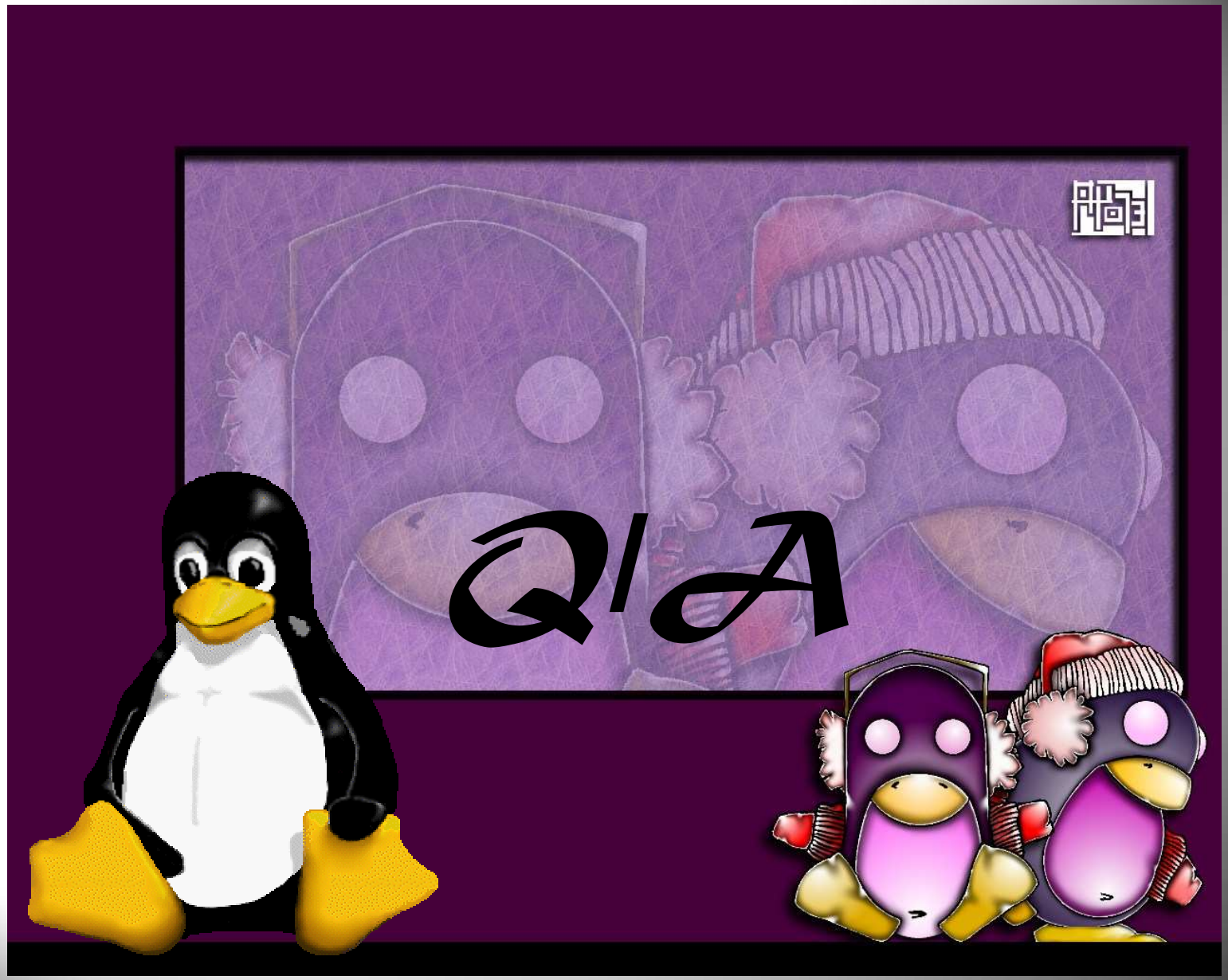
NPTL

- The new threading model has various advantages
- In practice, for good designed apps, the biggest one is the usage of futex to implement the pthread_mutex object
- The futex (unlike sched_yield) will avoid a scheduling collapse during heavy contentions of a lock among different threads
- NPTL is fully POSIX compiliant too (modulo RT)
- Due of the above point, NPTL **obviously** can't be enterely backwards compatible (not even at the source level) with linuxthreads

Clustering specific targets

- It would be nice to provide a standard process migration functionality in the future during 2.7
- The interconnects are getting faster and over time we might threat a cluster like we threat smp today
- Not all applications are ideal to be migrated transparently by the kernel, so for some clustering application there is no need of additional kernel support and it makes much more sense to scale the load all in userspace
- UML also provides interesting properties, but it introduces a significant cpu overhead

SuSE LINUX AG
THE LINUX EXPERTS



SuSE